Государственное образовательное учреждение
высшего профессионального образования
«Московский государственный технический университет
имени Н.Э. Баумана»

Отчет
По лабораторной работе №1
По курсу ««Конструирование компиляторов»
На тему
««Распознавание цепочек регулярного языка»

Студент: Мурашов И.Д.
Группа: ИУ7-23М
Вариант: 4
Преподаватель: Ступников А.А.

Москва, 2020

Оглавление

1 Цель и задачи работы

Цель работы: приобретение практических навыков реализации важнейших элементов лексических анализаторов на примере распознавания цепочек регулярного языка.
Задачи работы:

1. Ознакомиться с основными понятиями и определениями, лежащими в основе построения лексических анализаторов.

2. Прояснить связь между регулярным множеством, регулярным выражением, праволинейным языком, конечноавтоматным языком и недетерминированным конечноавтоматным языком.

3. Разработать, тестировать и отладить программу распознавания цепочек регулярного или праволинейного языка в соответствии с предложенным вариантом грамматики.

2 Текст программы

```
using Lab1_Regexp.Helpers;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Lab1_Regexp
{
public class Program
{
public static void Main(string[] args)
{
//List<char> alphabet = Enumerable.Range('a', 'z' - 'a' + 1).Select(i
List<char> alphabet = Enumerable.Range('a', 'b' - 'a' + 1).Select(i =
//alphabet = alphabet.Concat(Enumerable.Range('0', '9' - '0' + 1).Sel
GraphVizBuilder graphViz = new GraphVizBuilder();

// Read valid regexp
Console.WriteLine("Enter RegExp to build FA:");
string regexp = Console.ReadLine();

// Preprocess regexp (adding dots where concatenating)
string processedRegexp = Preprocess(regexp);

// Process regexp to postfix
string postfixRegexp = ToPostfix(processedRegexp);

// Building NFA in graph from postfix regexp
NFA nfa = ToNFA(postfixRegexp);
var nfaTable = nfa.BuildTable(epsAlphabet);  // Building NFA in table


// Draw NFA
string graphNFA = graphViz.BuildString(epsAlphabet, nfaTable, nfa.Ini
graphViz.CreateGraph(graphNFA, "nfa.png");

// Building DFA from NFA
DFA dfa = new DFA(nfaTable, alphabet, epsAlphabet.IndexOf(Translation




string graphDFA = graphViz.BuildString(alphabet, dfa.DFATable, dfa.In
graphViz.CreateGraph(graphDFA, "dfa.png");

//R

dfa.Reverse();
```

```
string graphDFAr = graphViz.BuildString(dfa.Alphabet, dfa.DFATable, d
graphViz.CreateGraph(graphDFAr, "dfa_r.png");

//RD
dfa.MakeDfa();

string graphDFArd = graphViz.BuildString(dfa.Alphabet, dfa.DFATable,
graphViz.CreateGraph(graphDFArd, "dfa_rd.png");

//RDR

dfa.Reverse();
string graphDFArdr = graphViz.BuildString(dfa.Alphabet, dfa.DFATable,
graphViz.CreateGraph(graphDFArdr, "dfa_rdr.png");

//RDRD
dfa.MakeDfa();
string graphDFArdrd = graphViz.BuildString(dfa.Alphabet, dfa.DFATable
graphViz.CreateGraph(graphDFArdrd, "dfa_rdrd.png");

while (true)
{
Console.WriteLine("Enter word to check allowness:");
string word = Console.ReadLine();
if (word == "‘")
break;

// Model FA for given word
bool res = dfa.Model(word, alphabet);
Console.WriteLine($"Result: {res}\n");
}
}

public static string Preprocess(string regexp)
{
StringBuilder nRegexp = new StringBuilder();

CharEnumerator c, up;
c = regexp.GetEnumerator();
up = regexp.GetEnumerator();

up.MoveNext();

while (up.MoveNext())
{
c.MoveNext();

nRegexp.Append(c.Current);

if ((char.IsLetterOrDigit(c.Current) || c.Current == ')' || c.Current
```

```
c.Current == '?' || c.Current == '+') && up.Current != ')' && up.Curr
up.Current != '*' && up.Current != '?' && up.Current != '+')
nRegexp.Append('.');
}

if (c.MoveNext())
nRegexp.Append(c.Current);

return nRegexp.ToString();
}

public static string ToPostfix(string regexp)
{
List<Operator> ops = new List<Operator>() { new Operator('*', 3), new
new Operator('.', 2), new Operator('|', 1), new Operator('(', 0), new

StringBuilder postfix = new StringBuilder();
Stack<char> operators = new Stack<char>();

CharEnumerator c = regexp.GetEnumerator();

while (c.MoveNext())
{
if (char.IsLetterOrDigit(c.Current))
postfix.Append(c.Current);
else if (c.Current == ops[5].Symbol)  // Open bracket
operators.Push(c.Current);
else if (c.Current == ops[6].Symbol)  // Close bracket
{
while (operators.Peek() != ops[5].Symbol)
postfix.Append(operators.Pop());

operators.Pop();
}
else if (ops.Any(op => op.Symbol == c.Current))
{
while (operators.Count > 0 &&
ops.Find(op => op.Symbol == operators.Peek()).Priority >= ops.Find(op
postfix.Append(operators.Pop());

operators.Push(c.Current);
}
}

while (operators.Count > 0)
postfix.Append(operators.Pop());

return postfix.ToString();
}
```

```
public static NFA ToNFA(string postfix)
{
Stack<NFA> nfas = new Stack<NFA>();

CharEnumerator c = postfix.GetEnumerator();

while (c.MoveNext())
{
if (char.IsLetterOrDigit(c.Current))
nfas.Push(new NFA(c.Current));
else if (c.Current == '.')
{
NFA b = nfas.Pop();
NFA a = nfas.Pop();
a.Concat(b);

nfas.Push(a);
}
else if (c.Current == '|')
{
NFA b = nfas.Pop();
NFA a = nfas.Pop();
a.Alter(b);

nfas.Push(a);
}
else if (c.Current == '*')
{
nfas.Peek().Star();
}
else if (c.Current == '+')
{
nfas.Peek().Plus();
}
}

return nfas.Pop();
}
}
}

using Lab1_Regexp.Helpers;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Lab1_Regexp
{
public class DFA
```

```
{
public List<List<List<int>>> DFATable { get; private set; }
public int Initial { get; set; }
public List<int> Finish { get; private set; }
public List<int> NonFinish { get; private set; }

public List<char> Alphabet { get; private set; }


public bool isTrueDfa { get; set; }

public DFA(List<List<List<int>>> nfaTable, List<char> alphabet, int e
{
this.MakeDfaFromNfa(nfaTable, alphabet, epsI, initState, finishState)
}

public void MakeDfaFromNfa(List<List<List<int>>> nfaTable, List<char>
{

Finish = new List<int>();
NonFinish = new List<int>();

Initial = 0;
isTrueDfa = true;
Alphabet = new List<char>(alphabet);
List<HashSet<int>> DStates = new List<HashSet<int>>();
Stack<HashSet<int>> stack = new Stack<HashSet<int>>();
Stack<int> stackIndexes = new Stack<int>();
List<List<int>> Dtran = new List<List<int>>();


var epsClosure0 = EpsClosure(new HashSet<int>() { initState }, nfaTab
DStates.Add(epsClosure0);
Dtran.Add(new List<int>(Enumerable.Repeat(-1, this.Alphabet.Count)));
stack.Push(epsClosure0);
stackIndexes.Push(0);


while (stack.Count > 0)
{
var state = stack.Pop();
var stateInd = stackIndexes.Pop();

for (int i = 0; i < this.Alphabet.Count; i++)
{
var nextStates = EpsClosure(Move(state, i, nfaTable), nfaTable, epsI)
int index = DStates.FindIndex(s => s.SetEquals(nextStates));

//for (int k = 0; k < DStates.Count ; k++)
//{
```

```
//     bool isSuperset = new HashSet<int>(DStates[k]).IsSupersetOf(nex
//     if (isSuperset)
//     {
//         index = DStates.IndexOf(state);
//         break;
//     }
//}

if (index == -1)
{
index = DStates.Count;
DStates.Add(nextStates);
Dtran.Add(new List<int>(Enumerable.Repeat(-1, this.Alphabet.Count)));
stack.Push(nextStates);
stackIndexes.Push(index);
}

Dtran[stateInd][i] = index;
}
}

DFATable = new List<List<List<int>>>(DStates.Count);
for (int i = 0; i < DStates.Count; i++)
{
DFATable.Add(new List<List<int>>(alphabet.Count));

if (DStates[i].Contains(finishState))
Finish.Add(i);
else
NonFinish.Add(i);

for (int j = 0; j < this.Alphabet.Count; j++)
{
DFATable[i].Add(new List<int>(1));
DFATable[i][j].Add(Dtran[i][j]);
}
}
}


public void MakeDfa()
{
if (this.Alphabet.Contains('eps'))
{
this.MakeDfaFromNfa(this.DFATable, this.Alphabet, this.Alphabet.Index
}
else
{
this.MakeDfaFromDfa(this.DFATable, this.Initial, this.Alphabet, this.
}
```

```
}



public void MakeDfaFromDfa(List<List<List<int>>> nfaTable, int initSt
{
Finish = new List<int>();
NonFinish = new List<int>();
Initial = 0;

List<HashSet<int>> DStates = new List<HashSet<int>>();
Stack<HashSet<int>> stack = new Stack<HashSet<int>>();
Stack<int> stackIndexes = new Stack<int>();
List<List<int>> Dtran = new List<List<int>>();


//DStates.Add(new HashSet<int>() { 0 });
DStates.Add(new HashSet<int>() { initState });
Dtran.Add(new List<int>(Enumerable.Repeat(-1, alphabet.Count)));
stack.Push(new HashSet<int>() { initState });
stackIndexes.Push(0);


while (stack.Count > 0)
{
var state = stack.Pop();
var stateInd = stackIndexes.Pop();

for (int i = 0; i < alphabet.Count; i++)
{
var nextStates = Move(state, i, nfaTable);
int index = DStates.FindIndex(s => s.SetEquals(nextStates));
if (index == -1)
{
index = DStates.Count;
DStates.Add(nextStates);
Dtran.Add(new List<int>(Enumerable.Repeat(-1, alphabet.Count)));
stack.Push(nextStates);
stackIndexes.Push(index);
}

Dtran[stateInd][i] = index;
}
}

DFATable = new List<List<List<int>>>(DStates.Count);
for (int i = 0; i < DStates.Count; i++)
{
DFATable.Add(new List<List<int>>(alphabet.Count));
```

```
if (DStates[i].Contains(finishState))
Finish.Add(i);
else
NonFinish.Add(i);

for (int j = 0; j < alphabet.Count; j++)
{
DFATable[i].Add(new List<int>(1));
DFATable[i][j].Add(Dtran[i][j]);
}
}

}


public void MakeOneFinalState()
{
// It converts dfa to nfa
if (this.Finish.Count > 1)
{
this.Alphabet = this.Alphabet.Concat(new char[] { 'eps' }.ToList()).T

// Adding eps trans to nothing for every state
for (int i = 0; i < this.DFATable.Count; i++)
{
this.DFATable[i].Add(new List<int>());
}

// Adding new state to direct  prev final states to this one
var newFinalState = new List<List<int>>();

// Adding empty trans
for (int i = 0; i < Alphabet.Count; i++)
newFinalState.Add(new List<int>());


DFATable.Add(newFinalState);


// Point prev final states to new state
for (int i = 0; i < this.Finish.Count; i++)
{
DFATable[this.Finish[i]][this.Alphabet.IndexOf(Translation.Eps)].Add(
}

this.Finish = new List<int>() { this.DFATable.IndexOf(newFinalState)
}

}
```

```
public void Reverse()
{
this.MakeOneFinalState();
var tableOps = new TableOps();
this.DFATable = tableOps.GetReverseFrom(this.DFATable);
var tmp = this.Finish[0];
this.Finish[0] = this.Initial;
this.Initial = tmp;

}

public void Minimize()
{
this.Reverse();
this.MakeDfa();
this.Reverse();
this.MakeDfa();
}

public bool Model(string word, List<char> alphabet)
{
if (word.Length == 0 && Finish.Contains(Initial))
return true;
else if (word.Length == 0 && !Finish.Contains(Initial))
return false;
else
{
CharEnumerator c = word.GetEnumerator();
c.MoveNext();
int letterI = alphabet.IndexOf(c.Current);
int nextState = DFATable[Initial][letterI][0];

while (c.MoveNext())
{
letterI = alphabet.IndexOf(c.Current);
nextState = DFATable[nextState][letterI][0];
}

if (Finish.Contains(nextState))
return true;
else
return false;
}
}

private HashSet<int> EpsClosure(HashSet<int> states, List<List<List<i
{
Stack<int> stack = new Stack<int>(states);
HashSet<int> epsClosure = new HashSet<int>(states);
```

```
while (stack.Count > 0)
{
int state = stack.Pop();
for (int i = 0; i < nfaTable[state][epsI].Count; i++)
{
int nextState = nfaTable[state][epsI][i];
if (!epsClosure.Contains(nextState))
{
epsClosure.Add(nextState);
stack.Push(nextState);
}
}
}

return epsClosure;
}

private HashSet<int> Move(HashSet<int> states, int letterI, List<List
{
HashSet<int> move = new HashSet<int>();


foreach (int state in states)
{
for (int i = 0; i < nfaTable[state][letterI].Count; i++)
{
int nextState = nfaTable[state][letterI][i];
move.Add(nextState);
}
}

return move;
}
}
}

using Lab1_Regexp.Helpers;
using System;
using System.Collections.Generic;
using System.Text;

namespace Lab1_Regexp
{
public class NFA
{
public State Initial { get; set; }
public State Finish { get; set; }
public int Size { get; set; }

public NFA(char symb)
```

```
{
Finish = new State(1, new List<Translation>());
Initial = new State(0, new List<Translation>() { new Translation(Fini
Size = 2;
}

public void Concat(NFA second)
{
Finish.Translations.Add(new Translation(second.Initial, Translation.E
Finish = second.Finish;

Size = UpdateStates();
}

public void Alter(NFA second)
{
State newFinish = new State(1, new List<Translation>());
Finish.Translations.Add(new Translation(newFinish, Translation.Eps));
second.Finish.Translations.Add(new Translation(newFinish, Translation
Finish = newFinish;
State oldInit = Initial;
Initial = new State(0, new List<Translation>() { new Translation(oldI
new Translation(second.Initial, Translation.Eps) });

Size = UpdateStates();
}

public void Star()
{
State newFinish = new State(1, new List<Translation>());
Finish.Translations.Add(new Translation(Initial, Translation.Eps, tru
Finish.Translations.Add(new Translation(newFinish, Translation.Eps));
Finish = newFinish;
State oldInit = Initial;
Initial = new State(0, new List<Translation>() { new Translation(oldI
new Translation(Finish, Translation.Eps) });

Size = UpdateStates();
}

public void Plus()
{
State newFinish = new State(1, new List<Translation>());  //
Finish.Translations.Add(new Translation(Initial, Translation.Eps, tru
Finish.Translations.Add(new Translation(newFinish, Translation.Eps));
Finish = newFinish;
State oldInit = Initial;
Initial = new State(0, new List<Translation>() { new Translation(oldI

Size = UpdateStates();
```

```
}

public List<List<List<int>>> BuildTable(List<char> alphabet)
{
List<List<List<int>>> table = new List<List<List<int>>>(Size);
for (int i = 0; i < Size; i++)
{
table.Add(new List<List<int>>(alphabet.Count));
for (int j = 0; j < alphabet.Count; j++)
table[i].Add(new List<int>());
}

table = Initial.BuildTable(table, alphabet);

return table;
}



public void PrintTable(List<List<List<int>>> table)
{
for (int i = 0; i < table.Count; i++)
{
for (int j = 0; i < table[i].Count; i++)
{
Console.WriteLine("State {0} goes by {1} to state {2}", i, j, table[i
}
}
}

private int UpdateStates()
{
return Initial.Update();
}
}
}

using System;
using System.Collections.Generic;
using System.Text;

namespace Lab1_Regexp.Helpers
{
class TableOps
{

public void PrintTransitions(List<List<List<int>>> table)
{
Console.WriteLine("Print table");
for (int x = 0; x < table.Count; x++)
```

```
{
for (int y = 0; y < table[x].Count; y++)
{
for (int z = 0; z < table[x][y].Count; z++)
{
Console.WriteLine("From {0} by {1} to {2}", x, y, table[x][y][z]);
}
}
}
}

public List<List<List<int>>> GetReverseFrom(List<List<List<int>>> ori
{
List<List<List<int>>> reverseTable = new List<List<List<int>>>(origin
for (int i = 0; i < original.Count; i++)
{
reverseTable.Add(new List<List<int>>(original[i].Count));
for (int j = 0; j < original[i].Count; j++)
reverseTable[i].Add(new List<int>());
}

for (int x = 0; x < original.Count; x++)
{
for (int y = 0; y < original[x].Count; y++)
{
for (int z = 0; z < original[x][y].Count; z++)
{
reverseTable[original[x][y][z]][y].Add(x);
}
}
}
return reverseTable;
}


public List<List<List<int>>> GetDfaFromNfa(List<List<List<int>>> orig
{
List<List<List<int>>> dfaTable= new List<List<List<int>>>(original.Co

List<List<int>> states = new List<List<int>>();

states.Add(new List<int> { 0 });
dfaTable.Add(original[0]);

for (int x = 0; x < dfaTable[0].Count; x++)
{
var tmp = new List<int>();

for (int y = 0; y < dfaTable[0][x].Count; y++)
{
```

```
tmp.Add(dfaTable[0][x][y]);
}

if (!states.Contains(tmp)) {
states.Add(tmp);
}

}

return dfaTable;
}

}
}
```

3 Проверка правильности программы

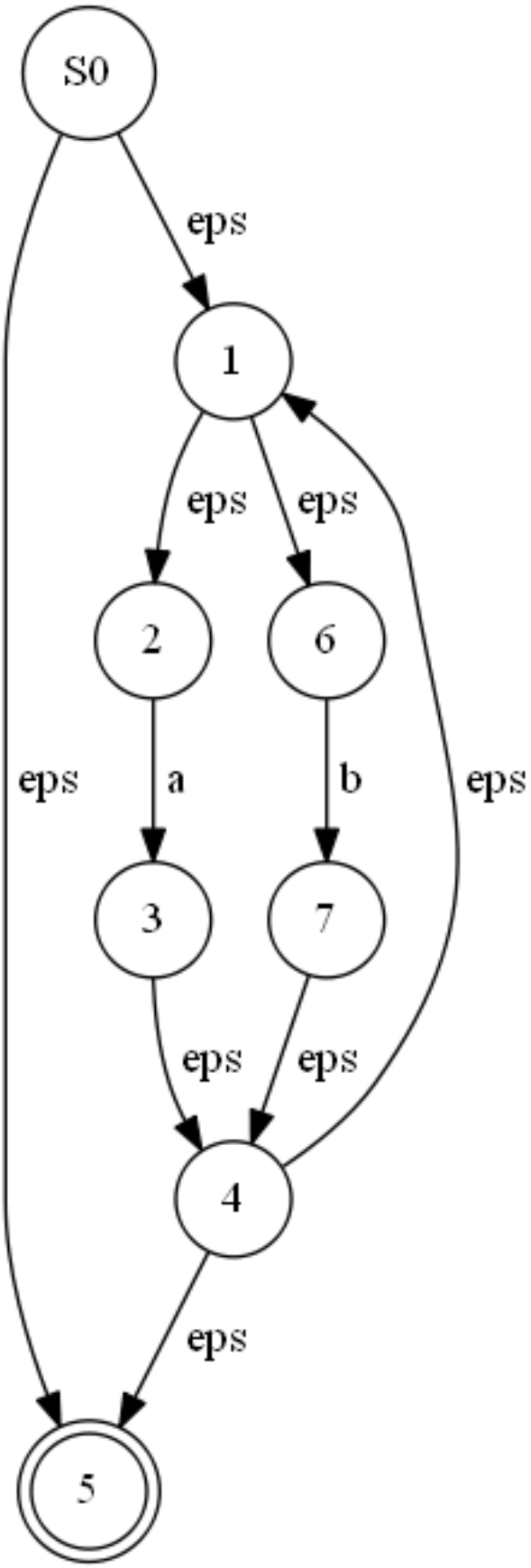
Все тесты производились для алфавита $\Sigma = \{a, b\}$. Начальные состояния на графе имеют префикс «S», а конечные выделяются двойным кругом. **Тестовые данные**:

Регулярное выражение	Входное слово	Ожидаемый результат
a	a	true
a	aa	false
a	b	false
ab	ab	true
ab	aba	false
ab	abb	false
ab	ba	false
(a|b)	a	true
(a|b)	b	true
(a|b)	ab	false
(a|b)	ba	false
a*	aaaaaaaa	true
a*		true
a*	b	false
(ab|b*)*	ab	true
(ab|b*)*	abb	true
(ab|b*)*		true
(ab|b*)*	aba	false
(ba|b*aa)+	baaa	true
(ba|b*aa)+		false
(ba|b*aa)+	babbaa	true

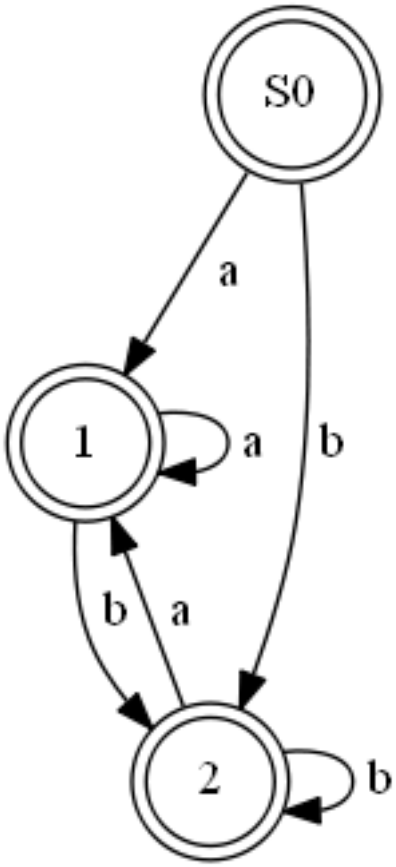
При тестировании работы программы результат работы программы для всех входных данных совпал с ожидаемым.

4 Пример работы программы

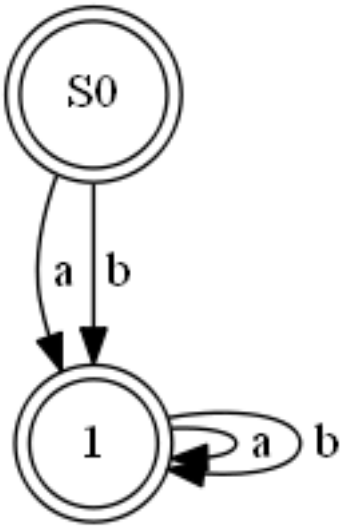
Выражение: (a|b)∗
Постфиксное регулярное выражение: ab|∗
НКА:



ДКА:



Минимальный ДКА:



5 Выводы

По результатам проведенной работы были приобретены практические навыки реализации важнейших элементов лексических анализаторов на примере распознавания цепочек регулярного языка. Было проведено ознакомление с основными определениями и понятиями, лежащими в основе построения лексических анализаторов. Была прояснена связь между регулярным множеством, регулярным выражением, праволинейным языком, конечноавтоматным языком и недетерминированным конечно-автоматным языком. Помимо этого

была разработана, оттестирована и отлажена программа распознавания цепочек регулярного или праволинейного языка в соответствии с предложенным вариантом грамматики. С использованием данной реализованной программы строятся допускающий данный язык НКА, ДКА и минимальный ДКА. Также моделируется работа минимального ДКА для проверки входной цепочки символов.

6 Список литературы

1. БЕЛОУСОВ А.И., ТКАЧЕВ С.Б. Дискретная математика: Учеб. Для вузов / Под ред. В.С. Зарубина, А.П. Крищенко. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2001.
2. АХО А.В, ЛАМ М.С., СЕТИ Р., УЛЬМАН Дж.Д. Компиляторы: принципы, технологии и инструменты. – М.: Вильямс, 2008.